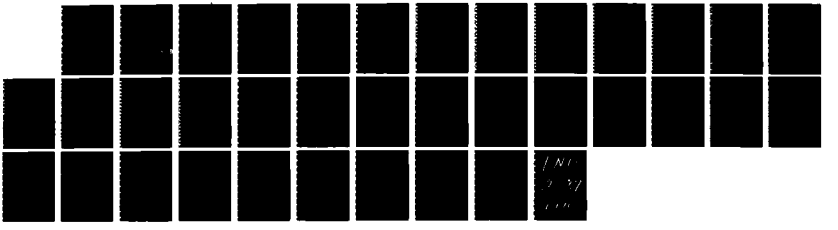
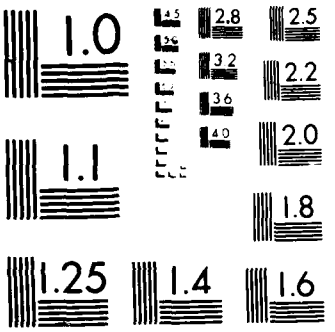
AD-A175 239 DEVELOPMENT OF PROGRAMMING TECHNIQUES FOR SIMD COMPUTERS(U) KENT STATE UNIV OH DEPT OF MATHEMATICAL SCIENCES J L POTTER 15 OCT 86 N00014-85-K-0010 1/1

UNCLASSIFIED F/G 9/2 NL





ROCOPY RESOLUTION TEST CHART

(12)

Development of Programming Techniques for SIMD Computers

Dr. J. L. Potter
Mathematical Sciences Department
Kent State University
Kent, Ohio 44242
(216) 672-2560

SECOND ANNUAL PROGRESS REPORT
N00014-85-K0010

October 15, 1986

86 12 16

CURRENT PROGRESS

During the past fiscal year, this research grant helped support activities in four related areas of associative programming: 1). implementation of the ASPROL compiler, 2). implementation of the FIRST, NEXT, PREVIOUS, TRUNC and TRUNCA acode functions and I/O stack routines, 3). implementation of the associative database primitives, and 4). analysis of new network designs for interprocessor communications in associative processors.

A copy of the ASPROL compiler has been given to Goodyear Aerospace for evaluation for use with their ASPRO SIMD computer. It is felt that this compiler can be augmented with rule based constructs that will provide a high performance expert system capability. Many of GAC's customers such as Grumman Aerospace, NOSC and NSWC are interested in such a capability. Preliminary studies performed under a seperate grant indicate that speeds of about 6000 inferences per second are obtainable.

The following papers were produced

DATA STRUCTURES FOR ASSOCIATIVE COMPUTING (submitted for publication)

AN ASSOCIATIVE MODEL OF DATA (submitted for publication)

ON PERFORMANCE OF SW-BANYAN NETWORKS (submitted for publication)

The ASPROL Programming Language.

The ASPROL language developed last year and described in last years report was implemented on the KSU VAX 780 research computer. The compiler converts ASPROL statements into a high level intermediate form similar to the form described in the

original proposal (p. 6). This intermediate code can be converted into assembly language code for any bit serial SIMD computer. The current implementation produces code for the STARAN E and the ASPRO computers. (Attached is a copy of the ASPROL users guide).

Primitive ACODE and I/O Stack Operators

The FIRST, NEXT, PREVIOUS, TRUNC and TRUNCA primitive ACODE operators developed for manipulating data structures during the first year of this effort were implemented and tested on the STARAN E. FIRST returns the acode of the next lower level in the hierarchy. NEXT returns the next acode at the current level of the hierarchy. PREVIOUS returns the previous acode at the current level. TRUNC returns the acode of the next higher level in the hierarchy. TRUNCA returns the acode of the top level of hierarchy.

Routines which simulate the I/O stack operations described in the original proposal were written. COLLAPSE eliminates the unused words in the I/O stack (a reserved page of the STARAN E array memory) compacting the data into a contiguous block. EXPAND reverses the process. It causes the contiguous block of data at the bottom of the stack to be expanded to words which are idle. The COLLAPSE and EXPAND commands simulate hardware commands that facilitate run time memory allocation in associative processors.

An Associative Database

An initial design for an associative database was developed and the primitive data base operators were implemented on the

STARAN E. This database is designed for natural language and artificial intelligence applications.

Interconnection Networks

A new method for determining the number of permutations realized by SW-Banyan networks was developed.

PLANS

The major emphasis in the next year will be to integrate the structure code mechanisms into the ASPROL compiler. As reported above, the primitive functions have been designed and implemented, but they have not been integrated into the ASPROL language. The first step will be to design the formal language mechanism which will be used in ASPROL and then the language will be expanded to include them.

The utility of the structure code concepts in associative computers and high level languages will be studies. Not only do structure codes allow dynamic memory allocation, they promise to provide a common data organization which will allow such diverse languages as LISP, FORTRAN, PROLOG and ASPROL to communicate with each other.

Some effort will be spent on the initial development of a theory of associative computing to explain its relationship with rule based languages.

APPENDIX A

ASPROL SYNTAX PRIMER
Version 2.00

ASPROL is a high level language designed specifically for SIMD parallel processors such as the STARAN-E and ASPRO. This primer is written to aid users in writing ASPROL codes. It specifies the correct syntax for writing ASPROL programs. It is not meant to be a user guide for writing STARAN-E programs. There is a class offered by the math department: Parallel Programming and a series of seminar about Parallel programming. For those interested in programming the STARAN-E, please consult with Dr. Jerry Potter.

For those who want to have a copy of this primer, use this command on your command level on Vax780 or Vax750 :

```
% lpr -1 /user/parallel/asprol/primer
```

I would appreciate your help debugging this compiler. Please send me mail about any bugs that you may find. Please be advised that ASPROL will quit compiling as soon as the first syntax error is detected. Please don't be discourage by this little inconvenience !

I. How to compile:

To compile your program :

```
% asprol -s filename.asp   (for STARAN-E code)
```
or
```
% asprol -a filename.asp   (for ASPRO code)
```

Either will create 3 files :

1. filename.lst :
   a readable program listing with intermediate code.

2. filename.iob :
   An encoded intermediate file which is expanded in pass two to assembly language.

3. Option -s will produce : filename.apl
   Assembler code which can be assembled by the STARAN-E assembler.

   Option -a will produce : filename.asm
   Assembler code which can be assembled by the ASPRO assembler.

II. Syntax :

```
[ Program identification ]
  Declaration
  Variable association
  body
[ end; ]
```

Note:

An "#include filename" may be placed anywhere in the source program for one level of file indirection. If the include file ends with an "endasm;" statement, it must be followed by at least two blank lines.

A. Program Identification:

option 1: Main Program_name
- Generate DFs and DC in assembler.
- Generate ENTRY's for DCs.

option 2: Subroutine Subroutine_name
- Generate DFs and ENTRYs for DC.

option 3: none
- Generate DFs and DC in assembler.
- Generate ENTRYs for DCs but no boiler plate.

example :

```
main program_name              subroutine routine_name
  .                              .
  variables association          variable association
  .                              .
  declarations                   declaration
  .                              .
  body                           body
  .                              .
end;                           end;
```

B. Declaration : Declaration should be specified in the following order !

1. define(var,constant);
   deflog(var,constant);
2. defvar
3. int scalar var1, var2, ...;
   int parallel var1, var2, ...;

   real scalar var1, var2, ...;
   real parallel var1, ...;

   index scalar var1, var2, ...;
   index parallel var1, var2, ...;

   logical scalar var1, var2, ...;
   logical parallel var1, var2, ...;

1. define/deflog

   define:

   syntax : define(var, constant);

   var : regular variable name.

   constant : any constant value (including a previous define).

   example : define(ten, 10);
             define(decade, ten);

deflog:

Deflog is used for defining a logical constant. In ASPROL, this definition is treated as a scalar constant. The only different between define and deflog is in its data type. Define is an integer scalar, while deflcg is a define logical constant (a special data type).

syntax : deflog(var, constant);

var : regular variable name.

constant : either 0 or 1.

example : deflog(true, 1);

2. defvar

If defvar is present in front of an int, real or logical declaration, the declared variables will be stored in the ASPROL symbol table and be treated as regular variables, but ASPROL will not cause any memory allocation macros to be generated. This option is provided to allow users to override the ASPROL compiler memory allocation so that they can allocate memory manually at assembly time. The programmer can specify his/her memory layout by using "asmcode" statement, or by editing the '.apl' file before assembling it.

example : defvar int parallel A:8[$], C[$];

3.1 int, real, index, logical

syntax: variable_name:field_width[dimension]

variable_name :

- letters : A - Z, a - z (no distinction)
  1 - 9
- Any variable name should begin with a letter. Any variable name longer than 6 characters will be truncated.

example : prog1, false, P12345.

3.2 field_width :

```
int     : 2-32 bits (scalar & parallel)
          (default 16 bits).
real    : 32 bits (scalar & parallel)
          (default).
index   : 1 bit scalar & parallel .
logical : 1 bit scalar & parallel .
```

Note:

All scalar variable are stored in 32 bit words regardless of declared size.

3.3 dimension:

The maximum limit for the dimension depends on the data type. For int, real, and logical data types, the maximum limit for the dimension is 3. The first dimension for the parallel mode should always be a '$'. The index data type is always 1 level deep.

Note :

Indexing of dimensioned arrays ranges from 0 to size - 1.

example :

```
int scalar A, B[32], C:8[10,10,10];
int parallel A[$], B[$,10,10], C:32[$],
    D:32[$,5,10];

real scalar A, B[10], C[10,10,10];
real parallel A[$], B[$,10], C[$,10,10];

index scalar A;
index parallel A[$];

logical scalar A;
logical parallel A[$], B[$,10], C[$,10,10],
                  XX[$];
```

C. Variable Association:

This section tells ASPROL which variables are associated with which variables. All variables used in this association should be declared in the previous declaration.

syntax : associate var1, var2, var3, ... with varX;

where : var1, var2,... are parallel mode variable.
varX is a logical parallel variable.

example : associate A[$], B[$], C[$], D[$] with XX[$];

D. Body :

1. If statement :

```
if (log_par_exp) then
     .
     body
     .
[else
     .
     body
     .]
endif;
```

2. For statement :

```
for index_var in (log_par_exp)
     .
     body
     .
endfor index_var;
```

3. Get statement:

```
get index_var in (log_par_exp)
     .
     body
     .
endget index_var;
```

4. Next statement:

```
next index_var in (log_par_exp)
     .
     body
     .
endnext index_var;
```

5. While statement:

```
while index_var in (log_par_exp)
     .
     body
     .
endwhile index;
```

6. Any statement:

```
any (log_par_exp)
     .
     body
     .
endany;
```

7. First statement:

```
first
 .
 initializations
 .
 loop statement
```

note :

Only simple assignment statements and initialization are allowed in the 'first' statement. The 'loop' statement should follow right after the 'first' statement.

8. Loop statement :

```
loop
 .
 body
 .
 until (scalar condition)
 .
endloop;
```

note :

There may be any number of 'until' statements in a 'loop -endloop' body.

9. Allocate statement:

```
allocate index in index_variable
 .
 body
 .
endallocate index;
```

note:

index_variable is index_variable association.
index is an index variable.

10. asmcode - endasm statement:

```
asmcode    (must be typed in the first column)
 .
 Staran-E assemble language instructions.
 .
endasm;    (must be typed in the first column)
```

note:

All statements between asmcode and endasm should exactly follow all rules for Staran-E assemble language. All ASPROL variables that are keywords in the STARAN-E assembly language should be prefixed with a '$' character.

example :

An x in ASPROL becomes $X in STARAN-E assembly language. Similarly, y becomes $Y and r6 becomes $R6.

For more STARAN-E keywords, please consult with the STARAN-E assemble language manual.

11. stop statement :

The 'Stop' statement is used to stop the execution of the program. It can be used anywhere in the program.

syntax : stop;

12 return statement :

The 'return' statement is used in the subroutine module only. It is used to return control of the execution to the instruction after the call statement in the calling module.

syntax : return;

13. call statement :

syntax : call subroutine_name;

14. setscope statement :

```
setscope log_par_exp
  .
 body
  .
endsetscope;
```

The setscope statement is used to set the M register temporarily with the log_par_exp result. The statements within the setscope - endsetscope only will be affected by the new M register setting. After the endsetscope statement, the M register will be restored to the value before the setscope statement is used.

15. unary function calls :

    1. MINDEX :

       syntax : MINDEX(par_var)

    2. MAXDEX :

       syntax : MAXDEX(par_var)

    3. MINVAL(par_var)

       syntax : MINVAL(par_var)

    4. MAXVAL(par_var)

       syntax : MAXVAL(par_var)

    MINDEX will return a index parallel variable that has set the responder bit which has the minimum value on the par_var array.

    MAXDEX will return a index parallel variable that has set the responder bit which has the maximum value on the par_var array.

    MINVAL will return a scalar temporary variable that has the minimum value of the par_var array.

    MAXVAL will return a scalar temporary variable that has the maximum value of the par_var array.

16. indirect address statement :

    syntax : @(scalar_exp):type:length[dimension]

    There are 2 different ways to use the indirect address statement :

    1. scalar result

       syntax :
       @(scalar_exp):type:length[index_par_var]

       This statement will return a scalar temporary variable which has the value in the array that is pointed to by the result of scalar_exp and indexed by the index_par_var.

```
example : @(A):int:8[xx]
where   : A is int scalar variable.
          xx is index__par_var.
```

2. parallel array result

```
syntax : @(scalar_exp):type:length[$]
```

This statement will return the array that is pointed to by the result of scalar_exp.

```
example : @(A):int:8[$]
where   : A is int scalar variable.
```

17. assignment statement :

```
log_par_var = log_par_exp;

var = expressions;
```

note :

All data types should have the same type for the left and right hand side.

arithmetic operators:

```
+ : addition
- : subtraction
* : multiplication
/ : division
```

relational operators:

```
<  , .lt.  : less than
>  , .gt.  : greater than
<= , .le.  : less than or equal
>= , .ge.  : greater than or equal
== , .eq.  : equal
!= , .ne.  : not equal
```

logical operators:

```
!  , .not. : negation
||  , .or. : or
&&  , .and.: and
^^  , .xor.: exclusive or
```

note :

If you use parallel variable, you should use $ in the assignment expression.

ex: var1[$] = var2[$] + var3[$] + 5;

NOTATIONS:

```
[..]           : optional
log_par_exp    : logical parallel expression
log_par_var    : logical parallel variable
index_par_var  : index parallel variable
```

III. Reserved keywords:

These following identifiers are reserved for use as keywords and may not be used otherwise:

int	real	scalar	parallel
index	logical	any	allocate
associate	define	deflog	defvar
during	end	endallocate	endany
endfor	endget	endif	endifnany
endloop	endnext	endsetscope	endwhile
else	elsenany	for	first
get	in	include	if
ifnany	loop	main	maxdex
maxval	mindex	minval	next
procedure	return	setscope	stop
subroutine	then	thenany	until
while	with	#include	

The above keywords which are not currently implemented by this compiler are reserved for future use. Some keywords may be changed for further refinement.

*** end ***

APPENDIX B

DATA STRUCTURES FOR ASSOCIATIVE SUPERCOMPUTERS

1.0 Introduction

1.1 Associative Super Computing

Associative Super Computing is a model of computation, where each datum has its own dedicated processor. Associative computing exploits massive fine grain parallelism in a natural way. Massively parallel associative computers with 16K processors have been built, eliminating the need for time sharing a single central processing unit with a multitude of data elements thus avoiding the classic memory - CPU bottleneck. Associative computing uses massive parallel searching in place of address calculation, reducing programming complexity. This paper describes a method for implementing data structures in the associative computing model which is applicable to all types of high level languages.

1.2 Background

The concept of associativity has been present in computer science for many years (Jacks,1971 and Findler, 1979). The most prevalent realization is associative triples in AI. The association list and ASSOC function in the LISP language are of course implemented in conventional hardware. Content addressable memories are manufactured but are limited in use due to their relatively high cost. Associative computers were first developed at Goodyear Aerospace in the early 1970s (Batcher, 1977). C. Foster's book (1976) describes the basic components of elementary associative computers.

The first higher level languages described that made use of

the parallel nature of associative computers were extensions of FORTRAN (Potter, 1982) and PASCAL (Reeves, 1985). However, more recent languages have been described which are based on an associative model of computation (e.g. Potter, 1987). Inherent in these languages is the concept that each association has its own dedicated processor and that computation is effected by repeatedly selecting associations to be processed. Since there is no formal mechanism in these languages to enable the specification of a data structure, associative data structures had to be implemented at the assembly language level (For example, Potter,1983, Reed,1985 and Potter,1985). This paper expands on the concepts used for assembly language data structures building a completely general hierarchy of data structures which can be used in any higher order language.

2.0 Associative Programming

2.1 Background

The thrust of the associative computing model can be best explained by analyzing the fundamental components of a program. In this section, a program is considered as a message between programmer and computer. The message contains two major types of information, the procedural component and the identification component. The procedural part specifies the operations to be performed and the order in which they are to be executed. The identification component of a program selects the data to be operated on by the procedural component. Associative computing uses associative addressing for the identification component.

The structure of data provides information. Thus in the

record shown in Figure 2.1, the number 50 can be recognized as the AGE attribute of the patient JONES not by any inherent property of its own, but by the fact that it is in the second (AGE) position of the record. College texts on data structures often describe information theory as developed by Shannon[1948] and others as part of the theoretical background for data structures. The knowledge that the second element of the data structure contains AGE data is "information" in the classical sense that it dispels "uncertainty" about the meaning of the number 50.

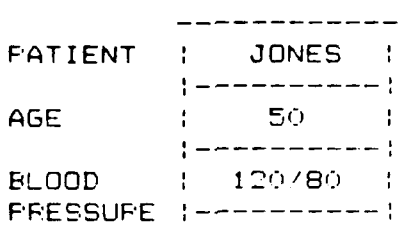


Figure 2.1 - A Simple Data Structure

The positional information content of a data structure is established by two mappings. The first mapping is between the problem data and the logical data structure based on the algorithm. The second mapping is between the logical data structure and the physical organization of the computer memory. These mappings are established by the programmer and are often the most crucial aspect of program development.

When a program is viewed as a message between a programmer and a computer, as illustrated in Figure 2.2, it contains two basic types of information. First, the program directs the computer in the procedural execution of the fundamental steps of

the algorithm. Second, the program embodies the positional information of the data structure by fetching and delivering to the central processing unit the appropriate data at the appropriate time. That is, if the algorithm is calculating hospital insurance premiums based on age, the program must deliver the age datum not the blood pressure datum to the CPU. Since this data delivery task involves the mapping relationships mentioned above, it can be modeled by an addressing function (AF) component of the program.

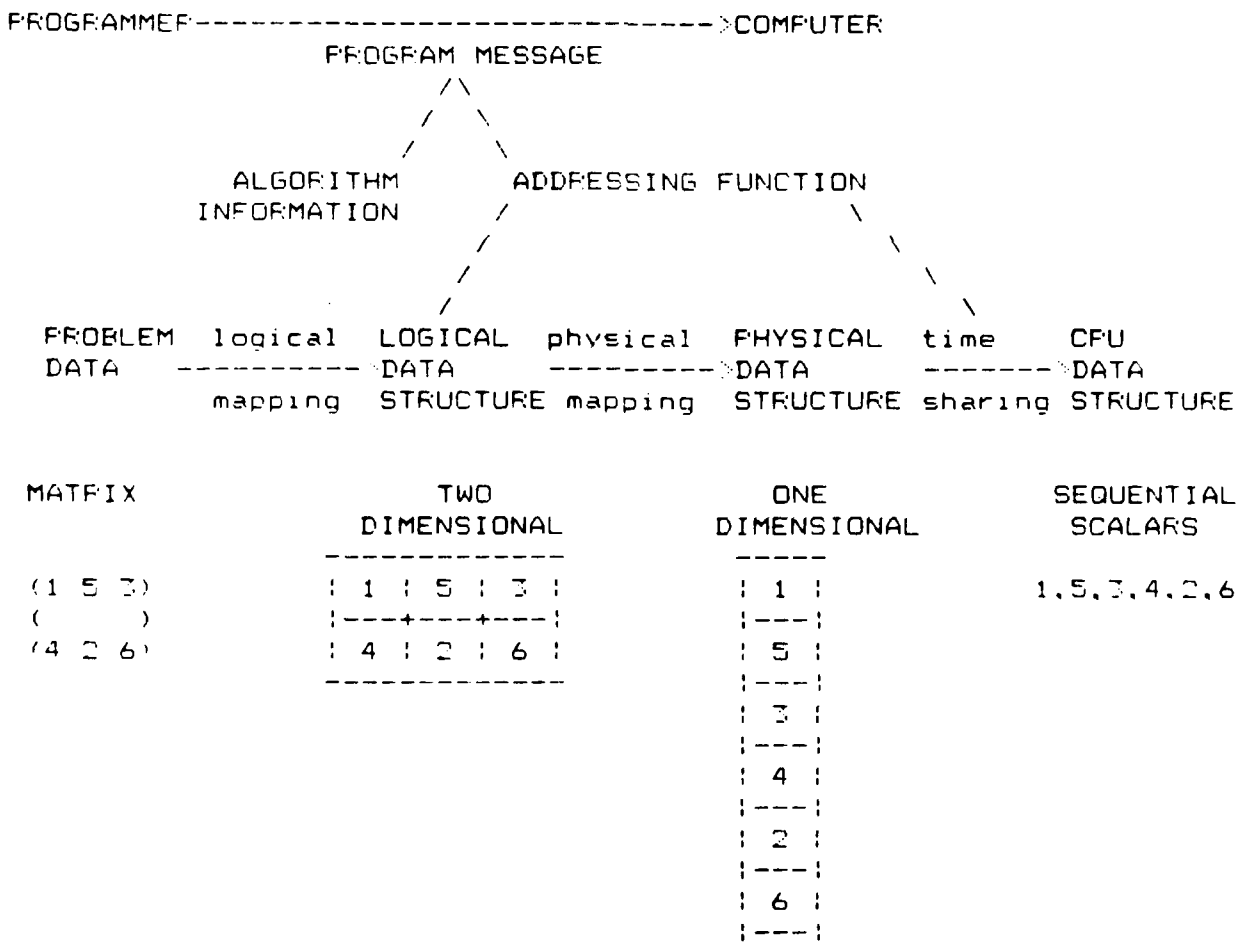


Figure 2.? A Conventional Program Message

The addressing function of every program is influenced by three factors: 1) the mapping between the problem data and the logical data structure (called the logical mapping), 2) the mapping between the logical data structure and the physical data structure in the computer memory (called the physical mapping), and 3) the mapping between the physical data structure and the CPU as a result of the fact that the CPU must be time shared among the multiple data records in a file.

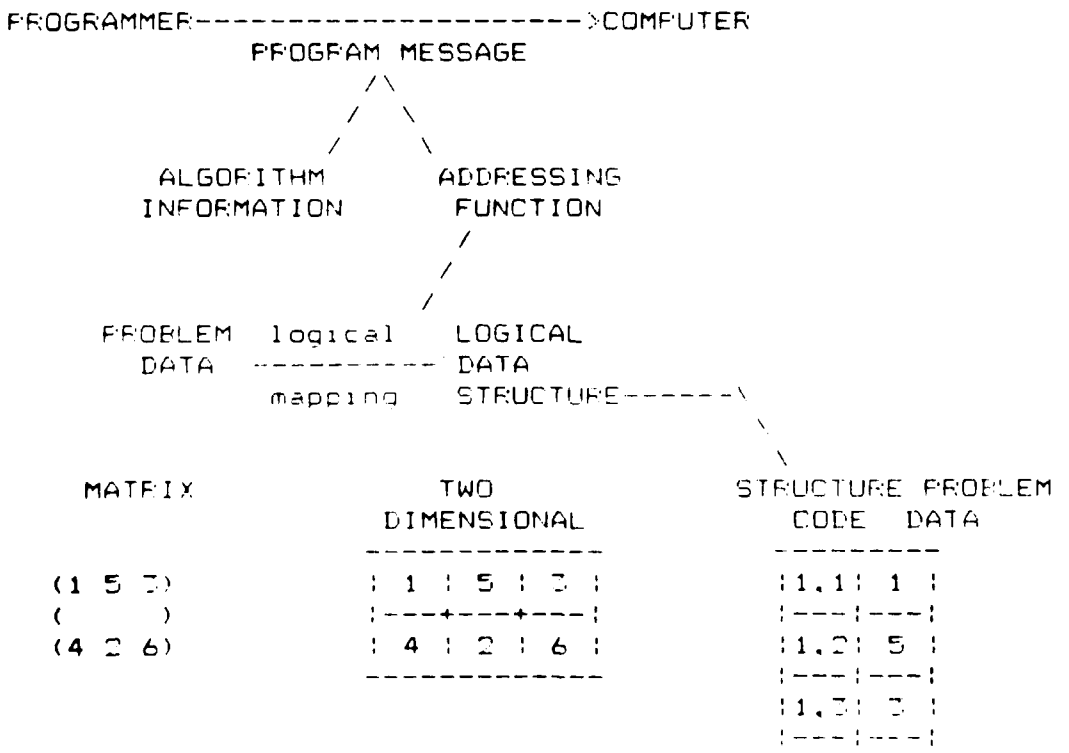
In the simplest conceptualization, a different addressing function is required for fetching each individual piece of data required by an algorithm. However, these simple addressing functions are combined into larger more comprehensive and complex functions using looping and address modification (indexing) techniques. The loop construct, for example, is used extensively to time share the CPU among the many identical records of a file. An important aspect of selecting a data structure for a sequential computer is to pick one which allows the addressing functions to be efficiently folded so that the loop construct can be used.

2.2 Associative Programming is Easy

Since associative computers reduce the complexity of addressing functions without recursion and without limiting the data structure, they are easier to program than conventional computers in three ways. First, since every data record has its own dedicated processor, the need for a "time sharing" factor in the

address function is eliminated. Second, the physical mapping component of the address function is replaced by parallel (associative) searching. Finally, as described in Section 3.0 the logical mapping relationship is stored associatively as Structure Codes with the data elements eliminating the need for run time calculations.

For example, in Figure 2.3, the logical portion of the address function consisting of the matrix row and column indices are stored with the data elements. Since the data structure codes are dependent only on the logical mapping, the programming task is reduced to 1) directing the computer in the sequential execution of the fundamental steps of the algorithm and 2) the manipulation of the logical data structure codes. The artifacts of time sharing the CPU and the physical sequential organization of memory are eliminated.



2,1	4
2,2	2
2,3	6

FIGURE 2.3 - An Associative Program Message

3.0 Associative Data Structures

3.1 Background

Arrays are the canonical forms of data structures. Their address functions form a natural hierarchy of complexity. Scalars are zero dimensional arrays. They are represented by the class of address functions consisting only of constants. The class of address functions for one dimensional arrays consist of constants plus one variable. Two dimensional arrays have two variable address functions, etc. The most common example of address functions for arrays, are the row-major and column-major ordering functions generated automatically for indexed arrays by most high order languages such as FORTRAN and PASCAL.

To date, content addressable memories (CAM) and associative computers (AC) have been used almost exclusively for storing and retrieving simple objects. An object is often defined as a location-value pair. Accordingly, an associative object contains two parts, a data element (or value) and the associated value of the logical addressing function (location) or Data Structure Code. The name of an object serves as its constant address function. As shown in Figure 3.1, simple scalar variables are synonymous with attribute value pairs in associative programming.

structure!data

code	element
attribute	value
age	50
size	large
color	blue
patient	jones

Figure 3.1 - Simple Scalar Structure Codes

3.2 Simultaneous Multiple Data Organizations

Since in the most general case, the value of an object can be of any composition (Elson,1973 p. 68). associative triples can also be accommodated by constant address functions. Note that three different sets of structure codes are illustrated in Figure 3.2. one for each part of the associative triple. Since all memory locations are searched in parallel, their is no a priori reason to select one set of structure codes over any other. In fact if there are n components in an object (i.e. values and codes) there are

$$\sum_{i=1}^{n-1} C_i^n$$

constant address functions. all of which can be used at the programmers discretion, intermingled in any order without any need for reordering. This is impossible in any sequentional computer since in a sequential computer, the data structures must be sorted to be efficiently accessed and data can be organized into only one ordering at a time.

3.3 Data Organization Manipulation.

The structure codes are discussed as if they were unique data items. In reality they are not. They are just like the

other data items in an associative object in that they can be searched for and manipulated. Structure codes are unique only in that they contain structural information on how one problem data element relates logically to the other problem data elements.

structure codes	problem	data

problem	data	structure codes

problem	structure codes	data

object	attribute	value
sofa	color	red
sofa	size	big
chair	color	blue

Figure 3.2 - Complex Scalar Structure Codes

3.4 Generalized Structure Codes

One dimensional arrays can be stored in CAM's and AC's by using a straight forward extension of scalar structure codes. The structure code consists of the object name (the constant portion of the address function) and the position of the value in the construct (the variable portion of the address function). The variable component for one dimensional arrays is simply the ordinal position of the data element in the array. Thus, for example, the one dimensional object A = (1 5 4 3 2) would have the structure code shown in Figure 3.3.

structure code			data
constant part	variable part		element
object name	element position		value
A	1		1
A	2		5
A	3		4
A	4		3
A	5		2

Figure 3.3 - A One Dimensional Array

The structure code for two dimensional arrays is a natural extension of one dimensional arrays as shown in Figure 3.4. The extension of structure codes to higher dimensional arrays is obvious. The composition and manipulation of these canonical array structure codes to make structure codes for complex compound data structure is considered next.

structure code			data
constant part	variable part		element
object name	row position	col	value
B	1	1	5
B	1	2	3
B	2	1	7
B	2	2	6

$$B = \begin{pmatrix} 5 & 3 \\ & \\ 7 & 6 \end{pmatrix}$$

Figure 3.4 - A Two Dimensional Array

4.0 Examples

4.1 Associative Data Structure References

One dimensional arrays are logical data structures which are

natural for use with several common problem data structures such as vectors, lists and strings. Two dimensional arrays are logical data structures which are natural for dealing with matrices and imagery. The mapping from these problem data structures to the logical data structure is the identity mapping. Consequently, for ease of reading, where no confusion can arise, the terms vector and matrix will be used interchangeably for one dimensional and two dimensional arrays respectively.

It is not uncommon to consider matrices as collections of vectors. Thus if the constant portion of the structure code shown in Figure 3.4 is modified to include "row position." The constant address "B 1" is shared by two values representing the vector (5 3) and "B 2" represents (7 6). Similarly, if the constant portion is modified to include "column position" instead of "row position," "B 1" represents $(5\ 7)^{-1}$ and "B 2" represents $(3\ 6)^{-1}$.

An important property of structure codes is the ability to reorganize them as illustrated above. The "." operator will be used to indicate the basis code grouping and can be thought of as a concatenation operator. The symbol, "?", is used as a place holder. Thus the code B.1.? represents the vector (5 3), B.?.2 represents $(3\ 6)^{-1}$, etc.

4.2 Data Structure Code Manipulation

The concept of combining data structures to form new data structures is common in some languages such a LISP. For example, lists can be grouped together to form lists of lists, etc. This can be done because of the generalized method of data storage for

lists. However, this capability can not be extended to other types of data structures such as arays because they use an addressing access function not a content addressable access function. In associative computing, the ability to create new data structures form existing data structures at run time is possible.

In order to describe how the structure codes for two arbitrary data structures can be combined to generate the structure codes for a combined data structure, several definitions are necessary. Let DS_j be a data structure of dimension r with address function A_j. Then $A_j = a_j0.a_j1..a_jr$ represents the r+1 components of the structure code. By convention, the 0th component is the constant portion which is the name of the data structure. Let $A_j(m)$ stand for the structure code of A_j for the mth element of DS_j. Let 0^1 denote the constant value 0, 0^2 denote 0.0, 0^3 denote 0.0.0, etc. Then 0^n denotes the constant zero structure code for a function with n components. Similarly, let $A^n(x)$ denote the first (left most) n components of a structure code. The depth of a component is equivalent to the number of components to its left.

Then if DSl is the complex data structure obtained by inserting data structure DSk with dimension s, as the mth element of DS_j with dimension r, at depth d, the address function Al for DSl has dimension d+s, and is given by

$$Al(x) = A_j(x).0^{d+s-r} \quad \text{for } x \neq m$$

$$Al(x) = A_j(x).^{d}Ak(y) \quad \text{for } x=m \text{ for all } y \text{ in DSk.}$$

The data structure insertion operation is denoted by:

receiving_data_structure!![element,depth]inserted_data_structure. If a complex structure is to be built by a number of insertions, they may occur in any order, i.e. if m1 != m2, then

(A1 !![m1,d1] Ai) !![m2,d2] Aj = (A1 !![m2,d2] Aj) !![m1,d1] Ai.

Figure 4.1 gives an example. DSa is an "empty" vector with address function Aa = (1 2). DSb and DSc are both matrices with the same address function Ab=Ac=(1.1, 1.2, 2.1, 2.2). The composition A=(Aa !![1,1] Ab) !![2,1] Ac is shown. Clearly, arbitrarily complex hierarchical data structures can be composed from the basic canonical forms.

This approach is completely general. Lists are simply a special case since they are "vectors" whose elements are atoms or other lists. Address function composition can be applied to list structure codes to generate the structure codes for any complex nested list.* Figure 4.2 illustrates the structure codes for a list.

* Structure codes for lists have been developed separately and described elsewhere (Potter,1983 and Reed,1985). The thrust of this paper is to develop a comprehensive approach to addressing functions and illustrate their generality. The reader is refered to the above references for details on programming using structure codes.

	Aa	Value
DSa	1	nil
DSa	2	nil

DSa = (nil nil)

	Ab	Value
DSb	1.1	7
DSb	1.2	14

DSb = (7 14)

DSb	2.1	3
DSb	2.2	8

	Ac	Value
DSc	1.1	0
DSc	1.2	15
DSc	2.1	6
DSc	2.2	2

DSb = (3 8)
(0 15)
(6 2)

OBJECT NAME	VECTOR POSITION	MATRIX ROW	MATRIX COL	VALUE
A	1	1	1	7
A	1	1	2	14
A	1	2	1	3
A	1	2	2	8
A	2	1	1	0
A	2	1	2	15
A	2	2	1	6
A	2	2	2	2

A = (((7 14) (0 15))
(( ) ( ))
((3 8) (6 2))

Figure 4.1 - A Vector of Matrices

OBJECT NAME	VECTOR POSITION	VECTOR POSITION	VECTOR POSITION	VALUE
LIST	1	0	0	THIS
LIST	2	1	0	IS
LIST	2	2	1	A
LIST	2	2	2	LIST

LIST = (THIS (IS (A LIST)))

Figure 4.2 - A List

4.3 Synonymous Data Structures

In some applications, it is desirable to view data structures in two or more ways. For example, a string can be thought of as a single variable containing a list of characters or as an array of characters. As can be seen in Figure 4.3, this

dual approach to referencing strings is a natural artifact of using associative addressing techniques. The string as a whole can be accessed by the structure code S." while the nth character in the string can be accessed by S.n. Note that this capability is due to the parallel associative implementation of structure codes and does not require multiple variable declarations or equivalences.

S = "A STRING"

OBJECT NAME	POSITION	VALUE
S	1	A
S	2	
S	3	S
S	4	T
S	5	R
S	6	I
S	7	N
S	8	G
S	9	null

Figure 4.3 - A String

4.4 Associative Stack and Queues

Other commonly used data structures, such as, stacks, queues, and linked storage can also be handled in the associative model. Stacks and queues are simply variable length vectors. A stack push is accomplished by adding a new (larger) ordinal position to the vector. A pop is simply the selection of the largest element of the vector and its removal. Queue and linked lists can likewise be easily implemented. However, it should be emphasized that these data structures are artifacts of conventional sequential structures, and that if the data to be stacked or queued is stored in associative memory with a time

tag, the need for these structures is eliminated.

5.0 Conclusions

This paper has presented a unified approach for representing arbitrarily complex data structures in content addressable memories and associative computers. This approach to data structures in associative computers has the advantages of 1) automatically extracting fine grain parallelism, 2) eliminating much of the complexity of the non-algorithmitic address computation in program development, 3) allowing multiple data structures to be associated with each datum, 4) allowing the data structures themselves to be modified, and 5) allowing information exchange between vastly different program languages such as LISP, PROLOG, OPS5, FORTRAN and PASCAL.

Some areas for future research are:

1) defining arithmetic operations on complex data structures as a natural extension of element by element arithmetic of vectors and matrices,

2) the utilization of multiple distinct structure codes in the same datum. In general, there can be a different structure code for every logical hierarchical data structure to which the datum belongs. This aspect may be particularly useful for semantic networks and frames in AI applications,

3) the development of universal operators for the manipulations of structure codes. For example, the operator "root" will generate the structure code for the root of a tree from the structure code of any of its nodes (See Potter,1985), and

4) the investigation of mathematical properties of addressing functions and structures codes.

6.0 Bibliography

[1]. Batcher, K. E., "Multidimensional Access Memory in STARAN," in IEEE COMPUTER, February, 1977, pp. 174-177.

[2]. Elson, Mark, "Concepts of Programming Languages," SRA, Chicago, 1973.

[3]. Findler, N. V. (ed.), "Associative Networks - Representation and Use of Knowledge by Computers," Academic Press, New York, 1979.

[4]. Foster, C. C., Content Addressable Parallel Processors, Van Nostrand Reinhold, 1976.

[5]. Jacks, E. L. (ed.), "Associative Information Techniques," Elsevier, New York, 1971.

[6]. Kohonen, T., Associative Memory: A system-theoretical approach," Springer-Verlag, Berlin, 1977.

[7]. Potter, Jerry L., "MPP Architecture and Software," NON-CONVENTIONAL COMPUTERS FOR IMAGE PROCESSING, L. Uhr and K. Preston (eds.), Academic Press, 1982.

[8]. Potter, Jerry L., "Alternative Data Structures for Lists in Parallel Associative Computers," in THE PROCEEDINGS OF THE 1983 ICPP, Bellaire, Michigan, August 23-26, 1983, pp.486-491.

[9]. Potter, Jerry L., "Specialized SIMD Instructions for Associative Processing," in PROCEEDINGS ON THE 1985 INTERNATIONAL CONFERENCE ON CIRCUIT DESIGN, Port Chester, New York, October 7-10, 1985, pp. 490-493.

[10]. Potter, Jerry L., "An Associative Model of Computation," Submitted to The Second International Conference on Super-Computing, May 4-7, 1987, San Francisco, Ca.

[11]. Reed, B. Jr., "An Implementation of Lisp on a SIMD Parallel Processor," in AEROSPACE APPLICATIONS OF AI, Dayton, Ohio, September 16-19, 1985.

[12]. Reeves, Anthony P., "Parallel Pascal and the Massively Parallel Processor," in THE MASSIVELY PARALLEL PROCESSOR, J. L. Potter, ed., pp. 230-260.

[13]. Shannon, C. E. and W. Weaver, "The Mathematical Theory of

Communication," University of Illinois Press, Urbana, 1949.

[14]. Tremblay, J. P. and P. G. Sorenson, "An Introduction to Data Structures with Applications," McGraw-Hill Book Co., N.Y., 1976.

END

2-87

DTIC